The IceCoffee Language Specification

Tel Aviv University*

The IC language is a simple object-oriented language that we will use in the compiler project. The goal is to build a complete compiler for this language and generate x86 assembly code. IC is essentially a subset of Java. It supports objects, subtyping, virtual and static methods, arrays; it is strongly-typed, and uses run-time checks to ensure the safety of object and array accesses; it supports dynamic allocation of objects and arrays, and uses an off-the-shelf garbage collector to reclaim heap memory. This document describes the structure and the semantics of IC.

1 Lexical Considerations

Identifiers and keywords are case-sensitive. Identifiers contain letters, digits, and the underscore character. Class identifiers start with an upper-case letter. All of the other identifiers start with a lower-case letter. The following are keywords and cannot be used as identifiers:

```
class   extends  static  void   int    boolean   string
return  if       else    while  break  continue  this
new     length   true    false  null
```

White spaces consist of space, tab, or newline characters. They may appear between any tokens. Keywords and identifiers must be separated by white space or a token that is neither a keyword or an identifier.

Both forms of Java comments are supported; a comment beginning with the characters `//` indicates that the remainder of the line is a comment. In addition, a comment can be a sequence of characters that begins with `/*`, followed by any characters, including newline, up to the first occurrence of the end sequence `*/`. Unclosed comments are lexical errors.

Integer literals may start with an optional negation sign `-`, followed a sequence of digits. Non-zero numbers should not have leading zeroes. Integers have 32-bit signed values between -2^{31} and $2^{31} - 1$.

*Course materials adapted with permission from Prof. Radu Rugina in Cornell university.

String literals are sequences of characters enclosed in double quotes. String characters can be: 1) printable ASCII characters (ASCII codes between decimal 32 and 126) other than quote " and backslash \, and 2) the escape sequences \" to denote quote, \\ to denote backslash, \t to denote tab, and \n for newline. No other characters or character sequences can occur in a string. Unclosed strings are lexical errors.

Boolean literals must be one of the keywords `true` or `false`. The only literal for heap references is `null`.

2 Program structure

A program consists of a sequence of class declarations. Each class in the program contains field and method declarations. A program must have exactly one method `main`, with the following signature:

```
static void main (string[] args) { ... }
```

Running the program with arguments `args` is equivalent to executing `A.main(args)`, where `A` is the class that contains the main method.

3 Variables

Program variables include local variables or method parameters, and are allocated on the run-time stack. Objects and arrays are allocated on the heap. Variables can hold integer and boolean values, or references to objects, arrays, or strings. Variables can be declared at any point in the program. They are visible from the declaration point up to the end of the innermost enclosing statement block.

Initialization. The program does not initialize variables with default values at declaration points. Instead, the compiler statically checks that each variable is assigned before it is used. Violations of this rule will cause compilation errors. On the other hand, object fields and array elements are initialized with default values (0 for integers, `false` for booleans, and `null` for references) when the objects or the arrays are created.

4 Strings

Unlike Java, IC has a primitive type `string`. Strings can be manipulated only in the following ways: using assignments of string references (including `null`); concatenating strings with the `+` operator; and testing for string reference equality using `==` (Note: this operator does not compare string contents). Library functions are provided to convert between strings and integer arrays containing the ascii codes of the characters in the string.

5 Arrays

The language supports arrays with arbitrary element types. If `T` is a type , then `T[]` is the type for and array with elements of type `T`. In particular, array elements can be arrays themselves, allowing programmers to build multidimensional arrays. For instance, the type `T[][]` describes a two-dimensional array constructed as an array of array references `T[]`.

Arrays are dynamically created using `new`, i.e., `new T[n]` allocates an array of type `T` with `n` elements and initializes the elements with their default values. The expression `new T[n]` yields reference to the newly created array. Arrays of size n are indexed from 0 to $n-1$ and the standard bracket notation is used to access array elements. If the expression `a` is a reference to an array of length n, then `a.length` is n, and `a[i]` returns the `(i+1)`-th element in the array. For each array access `a[i]`, the program checks at run-time that `a` is not null and that the access is within bounds: $0 \leq \texttt{i} < \texttt{n}$. Violations will terminate the program with an appropriate error message.

6 Classes

Classes are collections of fields and methods. They are defined using declarations of the form: "`class A extends B {` *body* `}`", where *body* is a sequence of field and method declarations. The `extends` clause is optional. When the extends clause is present, class `A` inherits all of the features (methods and fields) of class `B`. Only one class can be inherited, hence IC supports only single inheritance. We say that `A` is a subclass of `B`, and that `B` is a superclass of `A`.

Classes can only extend previously defined classes. In other words, a class cannot extend itself or another class defined later in the program. This ensures that the class hierarchy has a tree structure.

Method overloading is not supported. A class cannot have multiple methods with the same name, even if the methods have different number of types of arguments, or different return types. Hidden fields are not permitted either. All of the newly defined fields in a subclass must have different names than those in the superclasses.

However, methods can be overridden in subclasses. Subclasses can re-define more specialized versions of methods defined in their superclasses.

7 Subtyping

Inheritance induces a subtyping relation. When class `A` extends class `B`, `A` is a subtype of `B`, written $\texttt{A} \leq \texttt{B}$. Subtyping is also reflexive and transitive. Additionally, the special type `null` is a subtype of any reference type:

$$\frac{\texttt{A extends B } \{\ldots\}}{\texttt{A} \leq \texttt{B}} \qquad \frac{}{\texttt{A} \leq \texttt{A}} \qquad \frac{\texttt{A} \leq \texttt{B} \quad \texttt{B} \leq \texttt{C}}{\texttt{A} \leq \texttt{C}} \qquad \frac{}{\texttt{null} \leq \texttt{A}}$$

If `A` is a subtype of `B`, a value of type `A` can be used in the program whenever the program expects a value of type `B`.

Subtyping is not covariant for array types: if `A` is a subtype of `B` then `A[]` is *not* a subtype of `B[]`. Instead, array subtyping is type invariant, which means that each array type is only a subtype of itself.

8 Objects

Objects of a certain class can be dynamically created using the `new` construct. If `A` is a declared class, `new A()` allocates an object of class `A` on the heap and initializes all of its fields with the default values. The expression `new A()` yields a reference to the newly allocated object.

Object fields and instance methods are accessed using the '`.`' symbol. The expression `o.f` denotes the field `f` of object `o`, and the expression `o.m()` denotes a call to the virtual method `m` of object `o`. The keyword `this` refers to the current object (i.e., the object on which the current method is invoked).

Object references have class types: each definition `class A` introduces a class type `A`. Class types can then be used in declarations for reference variables. For instance, "`A obj`" declares a variable `obj` of type `A`, that is, a reference to an object of class `A`.

A class name `A` can be used as the type of an object reference anywhere in the program. In particular, it can appear in the body of the `class A` declaration itself, or even before that, as in the following example. This allows to build recursive and mutually recursive class structures, such as the ones below:

```
class List { int data; List next; }
class Node { int data; Edge[] edges; }
class Edge { int label; Node dest; }
```

9 Method Invocation

There are two kinds of method calls: static and virtual.

Static method calls are of the form $C.m(...)$, where C is a class name and m is a method declared static in C (or its superclasses). The program will execute this method, regardless of whether subclasses of C override this method or not. Hence, the method being executed is known at compile-time.

Virtual method calls are of the form $e.m(...)$, where e is an expression denoting an object reference. The method being executed is the method m of class C, where C is the *run-time* type of object e. Hence, the method executed is not known at compile-time. Virtual method calls are resolved at run-time via dynamic dispatch.

At each method invocation site, the program evaluates the actual arguments from left to right, and then assigns the computed values to the corresponding formal parameters of the method. Objects, arrays, and strings are passed by reference. The program then executes the body of the appropriate method, as described above. Upon return, the control is transferred

back to the caller. If the return statement has an expression argument, that argument is evaluated and the computed value is also returned to the caller.

At each method invocation, the number and types of actual values of the call site must be the same as the number and types of formal parameters in the method declaration. Furthermore, values returned by return statements must agree with the corresponding return types from method declarations. If a method is declared to return `void`, then return statements in the body of the method cannot return values. Such methods are allowed to reach the end of their body without a return statement. Otherwise, if the method is declared with a return type `T`, then all return statements must return values of type `T`. In this case, the method body is required to have a return statement on each program path. The compiler checks this requirement and reports an error whenever it is violated.

10 Scope rules

For each program, there is a hierarchy of scopes consisting of: the global scope, class scopes, method scopes, and local scopes for blocks within each method. The global scope consists of the names of all classes defined in the program. Each class has a static scope and an instance scope. The instance scope is the set of all fields and methods of that class; the static scope is the set of static methods only. The scopes of subclasses are nested inside the scopes of their superclasses. The scope of a method consists of the formal parameters and local variables defined in the block representing the body of the method. Finally, a block scope contains all of the variables defined in that block.

When resolving an identifier at a certain point in the program, the enclosing scopes are searched for that identifier. In particular, local variables and method parameters can only be used after they are defined in one of the enclosing block or method scopes. Fields and methods, both static and virtual, can be used either directly (without the dot prefix) if the current class contains those fields or method. The current class scope is the instance scope if the current method is virtual; or the static class scope if the current method is static. Fields and virtual methods can be used in expressions of the form *e.f* or *e.m()* when e has class type C and the instance scope of C contains those fields and methods. Finally, static methods can be used in expressions of the form $C.m()$ if the static scope of C contains m. Class names, fields, and methods can be used before the program declares them. However, the program must eventually declare them.

Identifiers, regardless of their kind, cannot be defined multiple times in the same scope. Here, *same* means exactly the same scope, not including the scopes it is nested in. The exception to this rule regards inheritance where identifiers, regardless of their kind, cannot be defined multiple times in the same scope or the scope of the superclass.

Therefore code like:

```
class A {
  int foo;
  void foo() {
  }
```

```
}
```

is illegal since 'foo' acts both as a field and a method name.

Code like this is legal:

```
class A {
  int x;
  void foo() {
     int x;
     x = 1; // here 'x' refers to the local variable 'x'
     this.x = 1; // here 'x' refers to the field 'x'
  }
}
```

Code like this is also legal:

```
class A {
   void foo() {
     int x;
     {
       boolean x;
       x = true; // 'x' refers to the variable defined in the inner scope.
     }
   }
}
```

Shadowing method parameters with local variables is illegal. For example:

```
class A {
   void foo(int x) {
     int x = 1;
   }
}
```

The following code is illegal since the same identifier (`foo`) appears both in the scope of the class `A` and in the scope of the class `B`, which inherits from `A`:

```
class A {
   int foo;
}
class B extends A {
   void foo() {}
}
```

11 Statements

IC has the standard control statements: assignments, method calls, `return` statements, `if` constructs, `while` loops, `break` and `continue` statements, and statement blocks.

Each assignment statement $l = e$ updates the location represented by l with the value of expression e. The updated location l can be a local variable, a parameter, a field, or an array element. The type of the updated location must match the type of the evaluated expression. For integers and booleans, the assignment copies the integer or boolean value. For string, array, or object types, the assignment only copies the reference.

Method invocations can be used as statements, regardless of whether they return values or not. In case the invoked method returns a value, that value is discarded.

The `if` statement has the standard semantics. It first evaluates the test expression, and executes one of its branches depending on whether the test is true of false. The else clause of an if statement always refers to the innermost enclosing if.

The `while` statement executes its body iteratively. At each iteration, it evaluates the test condition. If the condition is false, then it finishes the execution of the loop; otherwise it executes the loop body and continues with the next iteration. The `break` and `continue` statements must occur in the body of an enclosing loop in the current method. The `break` statement terminates the loop and the execution continues with the next statement after the loop body. The `continue` statement terminates the current loop iteration; the execution of the program proceeds to the next iteration and tests the loop condition. When `break` and `continue` statements occur in nested loops, they refer to the innermost loop.

Blocks of statements consist of a sequences of statements and variable declarations. Blocks are statements themselves, so blocks and statements can be nested arbitrarily deep.

12 Expressions

Program expressions include:

- memory locations: local variables, parameters, fields, or array elements;
- calls to methods with non-void return types;
- the current object `this`;
- new object or array instances, created with `new T()` or `new T [` e `]`;
- the array length expression e`.length`;
- unary or binary expressions; and
- integer, string, Boolean, and `null` literals.

13 Operators

Unary and binary operators include the following:

- Arithmetic operators: addition `+`, subtraction `-`, multiplication `*`, division `/`, and modulo `%`. The operands must be integers. The plus operator `+` is also used to concatenate strings. Division by zero and modulus of zero are dynamically checked, and cause program termination.
- Relational comparison operators: less than `<`, less or equal than `<=`, greater than `>`, and greater or equal then `>=`. Their operands must be integers.
- Equality comparison operators: equal `==` or different `!=`. The operands must have the same type. For integer and boolean types, operand values are compared. For the other types, references are compared.
- Conditional operators: short-circuit "and" `&&`, and short-circuit "or" `||`. If the first operand of `&&` evaluates to false, its second operand is not evaluated. Similarly, if the first operand of `||` evaluates to true, its second operand is not evaluated. The operands must be booleans.
- unary operators: sign change `-` for integers and logical negation `!` for booleans.

The operator precedence and associativity is defined by the table below. Here, priority 1 is the highest, and priority 9 is the lowest.

Priority	Operator	Description	Associativity
1	`[]` `()` `.`	array index, method call field/method access	left
2	`-` `!`	unary minus, logical negation	right
3	`*` `/` `%`	multiplication, division, remainder	left
4	`+` `-`	addition, subtraction	left
5	`<` `<=` `>` `>=`	relational operators	left
6	`==` `!=`	equality comparison	left
7	`&&`	short-circuit and	left
8	`||`	short-circuit or	left
9	`=`	assignment	right

14 IC Syntax

The language syntax is show in Figure 1. Here, keywords are shown using typewriter fonts (e.g., `while`); operators and punctuation symbols are shown using single quotes (e.g., '`;`'); the other terminals are written using small caps fonts (ID, CLASS, INTEGER, and STRING); and nonterminals using slanted fonts (e.g., *formals*). The remaining symbols are meta-characters: (...)* denotes the Kleene star operation and [...] denotes an optional sequence of symbols.

```
program     ::= classDecl*
classDecl   ::= class CLASS [extends CLASS] '{' (field | method)* '}'
field       ::= type ID (',' ID)* ';'
method      ::= [static] (type | void) ID '(' [formals] ')' '{' stmt* '}'
formals     ::= type ID (',' type ID)*
type        ::= int | boolean | string | CLASS | type '[' ']'

stmt        ::= location '=' expr ';'
              | call ';'
              | return [expr] ';'
              | if '(' expr ')' stmt [else stmt]
              | while '(' expr ')' stmt
              | break ';'
              | continue ';'
              | '{' stmt* '}'
              | type ID ['=' expr] ';'

expr        ::= location
              | call
              | this
              | new CLASS '(' ')'
              | new type '[' expr ']'
              | expr '.' length
              | expr binop expr
              | unop expr
              | literal
              | '(' expr ')'

call        ::= staticCall | virtualCall
staticCall  ::= CLASS '.' ID '(' [expr (',' expr)*] ')'
virtualCall ::= [expr '.'] ID '(' [expr (',' expr)*] ')'
location    ::= ID | expr '.' ID | expr '[' expr ']'

binop       ::= '+' | '-' | '*' | '/' | '%' | '&&' | '||'
              | '<' | '<=' | '>' | '>=' | '==' | '!='
unop        ::= '-' | '!'
literal     ::= INTEGER | STRING | true | false | null
```

Figure 1: IC Syntax

15 Typing rules

Typing rules for expressions

$$\frac{}{E \vdash \texttt{true} : \mathsf{bool}} \qquad \frac{}{E \vdash \texttt{false} : \mathsf{bool}}$$

$$\frac{}{E \vdash \textit{integer-literal} : \mathsf{int}} \qquad \frac{}{E \vdash \textit{string-literal} : \mathsf{string}}$$

$$\frac{E \vdash e_0 : \mathsf{int} \quad E \vdash e_1 : \mathsf{int} \quad op \in \{\texttt{+},\texttt{-},\texttt{/},\texttt{*},\texttt{\%}\}}{E \vdash e_0\ op\ e_1 : \mathsf{int}} \qquad \frac{E \vdash e_0 : \mathsf{string} \quad E \vdash e_1 : \mathsf{string}}{E \vdash e_0\ +\ e_1 : \mathsf{string}}$$

$$\frac{E \vdash e_0 : T_0 \quad E \vdash e_1 : T_1 \quad T_0 \leq T_1 \text{ or } T_1 \leq T_0 \quad op \in \{\texttt{==},\texttt{!=}\}}{E \vdash e_0\ op\ e_1 : \mathsf{bool}} \qquad \frac{E \vdash e_0 : \mathsf{int} \quad E \vdash e_1 : \mathsf{int} \quad op \in \{\texttt{<=},\texttt{<},\texttt{>=},\texttt{>}\}}{E \vdash e_0\ op\ e_1 : \mathsf{bool}}$$

$$\frac{E \vdash e_0 : \mathsf{bool} \quad E \vdash e_1 : \mathsf{bool} \quad op \in \{\texttt{\&\&},\texttt{||}\}}{E \vdash e_0\ op\ e_1 : \mathsf{bool}} \qquad \frac{E \vdash e : \mathsf{int}}{E \vdash \texttt{-}e : \mathsf{int}}$$

$$\frac{E \vdash e_0 : T\texttt{[]} \quad E \vdash e_1 : \mathsf{int}}{E \vdash e_0[e_1] : T} \qquad \frac{E \vdash e : \mathsf{bool}}{E \vdash \texttt{!}e : \mathsf{bool}}$$

$$\frac{E \vdash e : T\texttt{[]}}{E \vdash e\,\texttt{.length} : \mathsf{int}} \qquad \frac{E \vdash e : \mathsf{int}}{E \vdash \texttt{new}\ T[e] : T\texttt{[]}}$$

$$\frac{}{E \vdash \texttt{new}\ T() : T} \qquad \frac{}{E \vdash \texttt{null} : \mathsf{null}}$$

$$\frac{id : T \in E}{E \vdash id : T} \qquad \frac{E \vdash e : C \quad (id : T) \in C}{E \vdash e\,.\,id : T}$$

$$\frac{E \vdash e_0 : T_1 \times \ldots \times T_n \to T_r \quad E \vdash e_i : T_i' \quad T_i' \leq T_i \quad \text{for all } i = 1..n}{E \vdash e_0(e_1, \ldots, e_n) : T_r} \qquad \frac{(m : \textit{static}\ T_1 \times \ldots \times T_n \to T_r) \in C \quad E \vdash e_i : T_i' \quad T_i' \leq T_i \quad \text{for all } i = 1..n}{E \vdash C.m(e_1, \ldots, e_n) : T_r}$$

Typing rules for statements

$$\frac{E \vdash e_l : T \quad E \vdash e : T' \quad T' \leq T}{E \vdash e_l = e} \qquad \frac{E \vdash e : \mathsf{bool} \quad E \vdash S_1 \quad E \vdash S_2}{E \vdash \texttt{if}\ (e)\ S_1\ \texttt{else}\ S_2}$$

$$\frac{E \vdash e : \mathsf{bool} \quad E \vdash S_1}{E \vdash \texttt{if}\ (e)\ S_1} \qquad \frac{E \vdash e : \mathsf{bool} \quad E \vdash S}{E \vdash \texttt{while}\ (e)\ S} \qquad \frac{}{E \vdash \texttt{break}\ ;} \qquad \frac{}{E \vdash \texttt{continue}\ ;}$$

$$\frac{E \vdash e_0(e_1, ..., e_n) : T}{E \vdash e_0(e_1, ..., e_n)\ ;}\ [\textsc{Call}] \qquad \frac{E \vdash e : T \quad \texttt{ret} : T' \in E \quad T \leq T'}{E \vdash \texttt{return}\ e\ ;} \qquad \frac{\texttt{ret} : \mathsf{void} \in E}{E \vdash \texttt{return}\ ;}$$

$$\frac{E \vdash e : T' \quad T' \leq T \quad E,\, x : T \vdash S}{E \vdash T\ x = e\,;\ S}\ [\textsc{Decl}\ 1] \qquad \frac{E,\, x : T \vdash S}{E \vdash T\ x\,;\ S}\ [\textsc{Decl}\ 2] \qquad \frac{\begin{array}{c}S_1 \text{ not declaration} \\ E \vdash S_1 \quad E \vdash S_2\end{array}}{E \vdash S_1; S_2}$$

Rule Call says that a (virtual) method call, which is a legal expression is also legal as a statement. Rules Decl 1 and Decl 2 say that in order for a variable to be used in a statement it must be declared in a preceding statement.

Type-Checking Class and Method Declarations

$$\frac{\begin{array}{c}\mathit{classes}(P) = C_1, ..., C_n \\ \vdash C_i \ \text{ for all } i = 1..n\end{array}}{\vdash P}\ [\textsc{Program}]$$

$$\frac{\begin{array}{c}\mathit{methods}(C) = m_1, ..., m_k \\ \mathit{Env}(C, m_i) \vdash m_i \ \text{ for all } i = 1..k\end{array}}{\vdash C}\ [\textsc{Class}]$$

$$\frac{E, x_1 : t_1, ..., x_n : t_n, \texttt{ret} : t_r \vdash S_{body}}{E \vdash t_r\ m(t_1\ x_1, ..., t_n\ x_n)\ \{\ S_{body}\ \}}\ [\textsc{Method}]$$

Here, *Env(C,m)* is the environment (or scope) for class C and method m. If m is virtual, *Env(C,m)* is the instance scope of C and contains all of the methods and fields of C, including those declared in C's superclasses. If m is static, *Env(C,m)* is the static scope of C and contains only the static methods declared in C and its superclasses. Finally, *classes(C)* yields all of the classes in program P; and *methods(C)* yields the declarations of all methods in the body of class C (but not those inherited from other classes).

16 Library Functions

IC uses a simple mechanism to support I/O operations, datatype conversions, and other system-level functionality. The signatures of all of these functions are declared in a file `libic.sig`. The compiler reads this file and uses the declared signatures to type-check any uses of library functions. In this file, library function signatures are declared inside a class `Library`. Currently, the following methods are supported:

```
class Library {
 static void println(string s); /* prints string s followed by a newline */
 static void print(string s);   /* prints string s */
 static void printi(int i);     /* prints integer i */
 static void printb(boolean b); /* prints boolean b */

 static int readi();     /* reads one character from the input */
 static string readln(); /* reads one line from the input */
 static boolean eof();   /* checks end-of-file on standard input */

 static int stoi(string s, int n); /* returns the integer that s represents
                                      or n of s is not an integer */
 static string itos(int i);   /* returns a string representation of i */
 static int[] stoa(string s); /* an array with the ascii codes of chars in s */
 static string atos(int[] a); /* builds a string from the ascii codes in a */

 static int  random(int i); /* returns a random number between 0 and n-1 */
 static int  time();        /* number of milliseconds since program start */
 static void exit(int i);   /* terminates the program with exit code n */
}
```

All of the library methods are declared static, and are called accordingly, by qualifying the method call with the method name. For instance , `Library.random(100)` or `Library.stoi("412", -1)`. To generate the final executable, the assembly code generated by the compiler will be linked against library `libic.a`.

The grammar for the library signature file `libic.sig` is:

$$\begin{array}{rcl} \textit{libic} & ::= & \texttt{class Library } '\{' \ \textit{libmethod}^* \ '\}' \\ \textit{libmethod} & ::= & \texttt{static } (\textit{type} \,|\, \texttt{void}) \ \textsc{id} \ '(' \ [\textit{formals}] \ ')' \ ';' \end{array}$$

The default location of the library signature file `libic.sig` is the current directory. Alternatively, the location of this file can be explicitly specified in the command line using an argument of the form "`-L /path/to/libic.sig`".

This document specifies the LIR language and a simulator for the language. This version and the microLIR application were written by Roman Manevich.

1 LIR Language

A LIR program has the following format:

$$\{\text{StringLiteral} \mid \text{DispatchTable}\}^* \ \{\text{Instruction}\}^*$$

We now explain each of these elements.

1.1 String Literals

String literals have the format

```
name:  "text"
```

For example `str1: "In Foo_rise"`. The names of string literals can be used as addresses to integer array objects representing the string characters. These arrays are pre-allocated by the execution environment. Array names are constant global addresses; they can be accessed in any procedure but they cannot be assigned new values and the content of the strings cannot be modified (these are constant strings).

1.2 Dispatch Tables

Dispatch tables have the format

```
_name:  [labels]
```

where `_name` stores the address of the dispatch table, and `labels` is a comma-separated list of labels (labels are explained next), representing virtual functions that should be stored in the dispatch tables. As a convention, the name of the dispatch table of a class `_CLASS` is `_DV_CLASS`. The execution environment pre-allocates the table and fills the entries in the order specified with the addresses of the listed method labels. Dispatch table labels are constant global addresses that cannot be assigned new values, and the table contents cannot be modified.

1.2.1 Labels

Program labels match the pattern "`_[a-zA-Z_0-9]+:`". For example: `_this_is_a_label:`

Labels have addresses and can be used in implementing virtual function calls. Labels are constant global addresses; they can be accessed in any procedure but they cannot assigned new values.

1.3 Instructions

LIR uses two-operand instructions. As a general constraint – an instruction may only take one operand which involves a memory access. A summary of LIR instructions is shown in Table 1 and Table 2. In these tables, we use the term "Memory" to denote a name of a program variable (local variable or parameter), "Reg" to denote a LIR register, and "Immediate" to denote a constant value. Parameters stand for anything that indicates a value — a register (Reg), a variable (Memory), a constant (Immediate), a string name (indicating the string address), or a label (indicating its address). An immediate can be any 32-bit (signed) integer.

Register names should start with a capital R. Variables are legal (non-class) identifiers in IC. Registers and Memory names can be thought of as local variables; they can be accessed only in the procedure where they are first initialized (by an update instruction or if they are formal parameters of the procedure).

You can use a single `Move` instruction to move values between any combination of memory and register operands (other than memory-to-memory).

The LIR given here is a relaxation of the LIR shown in the text-book, which allows to perform some operations using a memory-operand. This is geared towards code-generation for an Intel platform.

The low-level instructions include: unary and binary instructions, copy instruction, array access instructions, length-of instruction, field access instructions, method calls, return instructions, labels, unconditional jumps, and conditional jumps branching on various conditions depending on the value resulting from the last `Compare` operation. The convention is that binary instructions of the form "OP a,b" mean b:=b OP a. You will also use a special instruction for library function calls (`Library`).

1.3.1 Runtime Checks Summary

"Fragile" instructions should be preceded by code to conduct runtime checks and code to handle erroneous situations. Your compiler is required to emit code to implement those checks and handle them by printing an error message and exiting properly. Both of these tasks can be done in either PA4 or PA5 (up to you). We suggest you consider implementing these checks only after you have finished all other tasks and made certain that your assignment is in good shape.

Operation	Runtime Check
a[i]	`__checkNullRef(a)` `__checkArrayAccess(a,i)`
a.length	`__checkNullRef(a)`
new T[n]	`__checkSize(n)`
o.f	`__checkNullRef(o)`
o.m()	`__checkNullRef(o)`
a / b	`__checkZero(b)`

The following are the textual error messages associated with the runtime checks:

```
Runtime Error: Null pointer dereference!
Runtime Error: Array index out of bounds!
Runtime Error: Array allocation with negative array size!
Runtime Error: Division by zero!
```

1.4 Comments

Lines beginning with # are comments. We encourage the use of comments to document where procedures begin and end and for documenting IC statements. For example:

```
# x = y.n
Move y,R1
MoveField R1.2,R2
Move R2,x
```

1.5 Dynamic Allocation

Dynamic allocation of objects and arrays is achieved via library functions.

Table 1: LIR Instruction Summary 1

Instruction	Op1	Op2	Meaning
Data Transfer Instructions			
`Move`	Immediate Reg Memory	Reg Memory	Move value between registers and memory, registers and registers, and immediate to register (memory to memory is illegal)
`MoveArray`	Reg[Reg] Reg[Immediate]	Reg	Array access (load)
	Reg Immediate	Reg[Reg] Reg[Immediate]	Array access (store)
`MoveField`	Reg.Reg Reg.Immediate	Reg	Field access (load)
	Reg Immediate	Reg.Reg Reg.Immediate	Field access (store)
`ArrayLength`	Memory Reg	Reg	Array/String length Op1 is array/string address Op2 stores the (returned) length
Arithmetic Instructions			
`Add`	Immediate Memory Reg	Reg	Add registers, memory and register, or immediate and register
`Sub`	Immediate Memory Reg	Reg	Subtract registers, memory and register, or immediate and register
`Mul`	Immediate Memory Reg	Reg	Multiply registers, memory and register, or immediate and register
`Div`	Immediate Memory Reg	Reg	Divide registers, memory and register, or immediate and register
`Mod`	Immediate Memory Reg	Reg	Remainder (modulus) registers, memory and register, or immediate and register
`Inc`	Reg		Increment register (by one)
`Dec`	Reg		Decrement register (by one)
`Neg`	Reg		Negate register
Logical Instructions			
`Not`	Reg		Bitwise negation of a register
`And`	Immediate Memory Reg	Reg	Bitwise-and of registers, memory and register, or immediate and register
`Or`	Immediate Memory Reg	Reg	Bitwise of registers, memory and register, or immediate and register
`Xor`	Immediate Memory Reg	Reg	Bitwise-xor of registers, memory and register, or immediate and register
`Compare`	Immediate Memory Reg	Reg	Compare values `Compare` = Op2-Op1

Table 2: LIR Instruction Summary 2

Instruction	Op1	Op2	Meaning
Control Transfer			
`Jump`	label		Unconditional jump
`JumpTrue`	label		Jump when true
`JumpFalse`	label		Jump when false
`JumpG`	label		Jump greater
`JumpGE`	label		Jump greater-or-equal
`JumpL`	label		Jump less than
`JumpLE`	label		Jump less-than-or-equal
`Library`	func-name(params)	Reg	Call library function and get return value in Reg
`StaticCall`	func-name({Memory=param}*)	Reg	Call static function and get return value in Reg
`VirtualCall`	Reg.Immediate({Memory=param}*)	Reg	Call virtual function and get return value in Reg
`Return`	param		Use the parameter as the returned value and return from call

Operation	Library Function	
new T()	`__allocateObject(s)`	s is an immediate representing the object size in bytes (fields + DVPTR)
new T[n]	`__allocateArray(s)`	s is an operand (immediate/register/memory) representing the number of array elements

The size of an object should be the number of fields plus 1 for the dispatch vector pointer. The value passed to `__allocateObject` should be multiplied by 4, since the function expects to receive the number of bytes required for the object.

Another important issue is updating the DVPTR field when a new object is allocated so that it is possible to perform calls to its methods. Here is an example of LIR instructions for allocating an object of class `A` with 2 fields:

```
# x = new A()
Library __allocateObject(12),R1
MoveField _DV_A,R1.0
Move R1,x
```

1.6 Procedures

Procedures are simply lists of LIR instructions preceded by a label. The label can be used by static calls and as an entry in a dispatch table for virtual calls. Execution starts from the procedure labeled `_ic_main`. There must be exactly one such label in a LIR program.

1.7 Procedure Calls and Returns

There are three types of procedure calls: virtual calls, static calls, and library calls:

Library Function Calls. Library functions include the functions in the `libic.sig` and the `stringCat` function. The library call instruction receives a comma-separated list of parameters that indicate the values of the arguments to be passed to the function and a register to receive the value optionally returned from the function. For example, `Library __println(str2),R2` receives the name of a string literal (its address) and prints the string literal at that address.

Static Function Calls. Calls to static functions consist of the function name, a comma-separated list of pairs, and a return value register. The list of pairs are of the form "Memory=param" where "Memory" stands for the name of the formal parameter and "param" is the actual parameter. For example, let `foo` be a static function in class `C`, declared as `static int foo(int x, boolean y)`. Then the call `w=C.foo(5,z)` can be translated into the LIR instructions

```
StaticCall _C_foo(x=5,y=z),R1
Move R1,w
```

Virtual Function Calls. Virtual function calls do not name the method directly. Instead the first register (left to the dot) indicates the address of the receiver object, and the second register (to the right of the dot) indicates the offset of the method in the object's dispatch table. The offset is given in terms of the number of integers — not the number of bytes. A detailed example of virtual function calls is found in the file `test_virtual_calls` under the `test` sub-directory of the microLIR application.

Function Returns. The LIR language specified here contains a `Return` function that uses a specified parameter as the actual returned value. A simplifying assumption we make is that every function can possibly return a value. In case of functions with a `void` return type, we suggest the following convention. The last instruction of function should be `Return 9999`. The exact value is not important but can be used to quickly identify (erroneous) situations where this value is used. When a function with `void` return type is called, a designated `Rdummy` register could be used as the last operand of the call instruction. The translation from LIR to assembly (in PA5) can safely ignore this register as an optimization.

1.7.1 Additional Library Functions

You will use an additional library function (will be supplied to you) to implement '+' between strings:

Function	Meaning
`__stringCat(s,t)`	concatenate strings s and t

2 microLIR Simulator

microLIR is a simulator for the LIR langauge. It is supposed to test your translation from IC to LIR before attempting to translate from LIR to the Intel assembly language. The simulator is a Java program that accepts a file containing a LIR program and executes the program.

2.1 Downloading and Installing the Simulator

The simulator can be downloaded from the course web site. Installation consists of simply unzipping the archive into your chosen installation directory. The installation contains Java sources, .class files, and several LIR sample programs (under the `test` sub-directory).

2.2 Program Usage

The program usage is:

```
Usage: file [options]
options:
  -printprog            Prints the program
  -verbose:num          Execution verbosity level (default=0)
                        0 - quiet mode
                        1 - announce next instruction to be executed
                        2 - announce instructions and computed values
  -stats                Prints program statistics and exits
```

In order to invoke the program enter `java -cp <LIR_INSTALLATION_PATH>/build microLIR.Main <args>` or add the `build` sub-directory to the CLASSPATH and enter `java microLIR.Main <args>`. The simulator supports all LIR instructions and most library functions. A batch file `microlir.bat` enables activating the program from a Windows environment.

2.3 Limitations

The simulator has undergone only very basic testing. If you believe you have found a bug, please send us the input causing the problem with a clear and detailed explanation. Here are some known limitations:

- The simulator handles only simple literal strings not containing escape characters.
- The simulator does not implement the following library functions: `readln`, `eof`, `stoa`, `atos`, and `random`.

2.4 Tips and Possible Pitfalls

Debugging An erroneous LIR program may cause behavior which may seem strange (just as your final translation may do when executed on an Intel machine). The simulator is equipped with functionality to detect various problems (access to illegal addresses, division by zero, allocating objects with sizes that are not multiples of 4, etc.). However, not all problems are detected. In order to understand what is wrong in your LIR program, you may raise the verbosity level of the execution and follow the list of instructions executed and the values computed by the instructions..

`main` fall-through The simulator processes the instructions one after another, unaware of procedure boundaries. Therefore, if the `_ic_main` procedure is not the last procedure, the simulator may follow by executing other code, not as you intended. There are two possible solutions for this: (i) print the translation of main as the last part of the program, or (ii) add a `Library __exit(0),Rdummy` instruction as the last instruction in the main procedure.

Missing `Return` instructions Every procedure, **except** `_ic_main`, should end with a `Return` instruction. A missing `Return` instruction results in following the execution with the first instruction of the procedure below.

Addresses The simulator allocates addresses for labels using their position in the sequence of instructions. Addresses of allocated objects (including strings and dispatch tables) start from $30,000$. The simulator never releases memory, and thus the address of an allocated object is never returned in subsequent allocations.

`Compare` register The Intel architecture includes an `EFLAGS` register, which behaves similarly to the `Compare` register of microLIR. However, while the `Compare` register only updates on `Compare` instructions, the `EFLAGS` register updates on every arithmetic instruction. Therefore, your LIR program should not depend on the value of the `Compare` register after an arithmetic instruction is executed. Instead, you should aim to use the value of the `Compare` register (by one of the `JumpXX` instructions) immediately after a `Compare` instruction.

Variable Shaddowing. IC allows local variables to be hidden by defining variables with the same name in inner blocks. Therefore, each variable should be given a unique name to avoid name clashes. A possible naming scheme is the following. Local variables names can be `vIDname` where `ID` is a unique id number and `name` is the actual name. Similarly, parameter names can follow the pattern `pIDname` and field names can follow the pattern `fIDname` etc.

`--nomin`
skip the DFA minimisation step during scanner generation.

`--jlex`
tries even harder to comply to JLex interpretation of specs.

`--dot`
generate graphviz dot files for the NFA, DFA and minimised DFA. This feature is still in alpha status, and not fully implemented yet.

`--dump`
display transition tables of NFA, initial DFA, and minimised DFA

`--legacydot`
dot (`.`) meta character matches `[^\n]` instead of `[^\n\r\u000B\u000C\u0085\u2028\u2029]`

`--noinputstreamctor`
don't include an InputStream constructor in the generated scanner

`--verbose` or `-v`
display generation progress messages (enabled by default)

`--quiet` or `-q`
display error messages only (no chatter about what JFlex is currently doing)

`--time`
display time statistics about the code generation process (not very accurate)

`--version`
print version number

`--info`
print system and JDK information (useful if you'd like to report a problem)

`--unicodever <ver>`
print all supported properties for Unicode version `<ver>`

`--help` or `-h`
print a help message explaining options and usage of JFlex.

3 A simple Example: How to work with JFlex

To demonstrate what a lexical specification with JFlex looks like, this section presents a part of the specification for the Java language. The example does not describe the whole lexical structure of Java programs, but only a small and simplified part of it (some keywords, some operators, comments and only two kinds of literals). It also shows how to interface with the LALR parser generator CUP [8] and therefore uses a class `sym` (generated by CUP), where integer constants for the terminal tokens of the CUP grammar are declared. JFlex comes with a directory `examples`, where you can find a small standalone scanner that doesn't need other tools like CUP to give you a running example. The "`examples`" directory also contains a *complete* JFlex specification of the lexical structure of Java programs together with the CUP parser specification for Java by C. Scott Ananian, obtained from the CUP [8] web site

(it was modified to interface with the JFlex scanner). Both specifications adhere to the Java Language Specification [7].

```
/* JFlex example: part of Java language lexer specification */
import java_cup.runtime.*;

/**
 * This class is a simple example lexer.
 */
%%


%class Lexer
%unicode
%cup
%line
%column


%{
  StringBuffer string = new StringBuffer();

  private Symbol symbol(int type) {
    return new Symbol(type, yyline, yycolumn);
  }
  private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline, yycolumn, value);
  }
%}


LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]
WhiteSpace     = {LineTerminator} | [ \t\f]

/* comments */
Comment = {TraditionalComment} | {EndOfLineComment} | {DocumentationComment}

TraditionalComment   = "/*" [^*] ~"*/" | "/*" "*"+ "/"
// Comment can be the last line of the file, without line terminator.
EndOfLineComment     = "//" {InputCharacter}* {LineTerminator}?
DocumentationComment = "/**" {CommentContent} "*"+ "/"
CommentContent       = ( [^*] | \*+ [^/*] )*

Identifier = [:jletter:] [:jletterdigit:]*

DecIntegerLiteral = 0 | [1-9][0-9]*


%state STRING

%%
```

```
/* keywords */
<YYINITIAL> "abstract"          { return symbol(sym.ABSTRACT); }
<YYINITIAL> "boolean"           { return symbol(sym.BOOLEAN); }
<YYINITIAL> "break"             { return symbol(sym.BREAK); }


<YYINITIAL> {
  /* identifiers */
  {Identifier}                  { return symbol(sym.IDENTIFIER); }

  /* literals */
  {DecIntegerLiteral}           { return symbol(sym.INTEGER_LITERAL); }
  \"                            { string.setLength(0); yybegin(STRING); }

  /* operators */
  "="                           { return symbol(sym.EQ); }
  "=="                          { return symbol(sym.EQEQ); }
  "+"                           { return symbol(sym.PLUS); }

  /* comments */
  {Comment}                     { /* ignore */ }

  /* whitespace */
  {WhiteSpace}                  { /* ignore */ }
}


<STRING> {
  \"                            { yybegin(YYINITIAL);
                                  return symbol(sym.STRING_LITERAL,
                                  string.toString()); }
  [^\n\r\"\\]+                  { string.append( yytext() ); }
  \\t                           { string.append('\t'); }
  \\n                           { string.append('\n'); }

  \\r                           { string.append('\r'); }
  \\\"                          { string.append('\"'); }
  \\                            { string.append('\\'); }
}


/* error fallback */
[^]                             { throw new Error("Illegal character <"+
                                                  yytext()+">"); }
```

From this specification JFlex generates a `.java` file with one class that contains code for the scanner. The class will have a constructor taking a `java.io.Reader` from which the input is read. The class will also have a function `yylex()` that runs the scanner and that can be used to get the next token from the input (in this example the function actually has the name `next_token()` because the specification uses the `%cup` switch).

As with JLex, the specification consists of three parts, divided by `%%`:

- usercode,
- options and declarations and
- lexical rules.

3.1 Code to include

Let's take a look at the first section, "user code": The text up to the first line starting with `%%` is copied verbatim to the top of the generated lexer class (before the actual class declaration). Beside `package` and `import` statements there is usually not much to do here. If the code ends with a javadoc class comment, the generated class will get this comment, if not, JFlex will generate one automatically.

3.2 Options and Macros

The second section "options and declarations" is more interesting. It consists of a set of options, code that is included inside the generated scanner class, lexical states and macro declarations. Each JFlex option must begin a line of the specification and starts with a `%`. In our example the following options are used:

- `%class Lexer` tells JFlex to give the generated class the name "Lexer" and to write the code to a file "`Lexer.java`".
- `%unicode` defines the set of characters the scanner will work on. For scanning text files, `%unicode` should always be used. The Unicode version may be specified, e.g. `%unicode 4.1`. If no version is specified, the most recent supported Unicode version will be used - in JFlex 1.6, this is Unicode 7.0. See also section 5 for more information on character sets, encodings, and scanning text vs. binary files.
- `%cup` switches to CUP compatibility mode to interface with a CUP generated parser.
- `%line` switches line counting on (the current line number can be accessed via the variable `yyline`)
- `%column` switches column counting on (current column is accessed via `yycolumn`)

The code included in `%{...%}` is copied verbatim into the generated lexer class source. Here you can declare member variables and functions that are used inside scanner actions. In our example we declare a `StringBuffer` "`string`" in which we will store parts of string literals and two helper functions "`symbol`" that create `java_cup.runtime.Symbol` objects with position information of the current token (see section 9.1 *JFlex and CUP* for how to interface with the parser generator CUP). As JFlex options, both `%{` and `%}` must begin a line.

The specification continues with macro declarations. Macros are abbreviations for regular expressions, used to make lexical specifications easier to read and understand. A macro declaration consists of a macro identifier followed by `=`, then followed by the regular expression it represents. This regular expression may itself contain macro usages. Although this allows a grammar like specification style, macros are still just abbreviations and not non terminals – they cannot be recursive or mutually recursive. Cycles in macro definitions are detected and reported at generation time by JFlex.

Here some of the example macros in more detail:

- `LineTerminator` stands for the regular expression that matches an ASCII CR, an ASCII LF or an CR followed by LF.
- `InputCharacter` stands for all characters that are not a CR or LF.
- `TraditionalComment` is the expression that matches the string `"/*"` followed by a character that is not a `*`, followed by anything that does not contain, but ends in `"*/"`. As this would not match comments like `/****/`, we add `"/*"` followed by an arbitrary number (at least one) of `"*"` followed by the closing `"/"`. This is not the only, but one of the simpler expressions matching non-nesting Java comments. It is tempting to just write something like the expression `"/*" .* "*/"`, but this would match more than we want. It would for instance match the whole of `/* */ x = 0; /* */`, instead of two comments and four real tokens. See DocumentationComment and CommentContent for an alternative.
- `CommentContent` matches zero or more occurrences of any character except a `*` or any number of `*` followed by a character that is not a `/`
- `Identifier` matches each string that starts with a character of class `jletter` followed by zero or more characters of class `jletterdigit`. `jletter` and `jletterdigit` are predefined character classes. `jletter` includes all characters for which the Java function `Character.isJavaIdentifierStart` returns `true` and `jletterdigit` all characters for that `Character.isJavaIdentifierPart` returns `true`.

The last part of the second section in our lexical specification is a lexical state declaration: `%state STRING` declares a lexical state `STRING` that can be used in the "lexical rules" part of the specification. A state declaration is a line starting with `%state` followed by a space or comma separated list of state identifiers. There can be more than one line starting with `%state`.

3.3 Rules and Actions

The "lexical rules" section of a JFlex specification contains regular expressions and actions (Java code) that are executed when the scanner matches the associated regular expression. As the scanner reads its input, it keeps track of all regular expressions and activates the action of the expression that has the longest match. Our specification above for instance would with input "`breaker`" match the regular expression for `Identifier` and not the keyword "`break`" followed by the Identifier "`er`", because rule `{Identifier}` matches more of this input at once (i.e. it matches all of it) than any other rule in the specification. If two regular expressions both have the longest match for a certain input, the scanner chooses the action of the expression that appears first in the specification. In that way, we get for input "`break`" the keyword "`break`" and not an Identifier "`break`".

Additional to regular expression matches, one can use lexical states to refine a specification. A lexical state acts like a start condition. If the scanner is in lexical state `STRING`, only expressions that are preceded by the start condition `<STRING>` can be matched. A start condition of a regular expression can contain more than one lexical state. It is then matched when the lexer is in any of these lexical states. The lexical state `YYINITIAL` is predefined and is also the state in which the lexer begins scanning. If a regular expression has no start

conditions it is matched in *all* lexical states.

Since you often have a bunch of expressions with the same start conditions, JFlex allows the same abbreviation as the Unix tool `flex`:

```
<STRING> {
  expr1   { action1 }
  expr2   { action2 }
}
```

means that both `expr1` and `expr2` have start condition `<STRING>`.

The first three rules in our example demonstrate the syntax of a regular expression preceded by the start condition `<YYINITIAL>`.

```
<YYINITIAL> "abstract"  { return symbol(sym.ABSTRACT); }
```

matches the input "`abstract`" only if the scanner is in its start state "`YYINITIAL`". When the string "`abstract`" is matched, the scanner function returns the CUP symbol `sym.ABSTRACT`. If an action does not return a value, the scanning process is resumed immediately after executing the action.

The rules enclosed in

```
<YYINITIAL> {
...
}
```

demonstrate the abbreviated syntax and are also only matched in state `YYINITIAL`.

Of these rules, one may be of special interest:

```
\"  {  string.setLength(0); yybegin(STRING); }
```

If the scanner matches a double quote in state `YYINITIAL` we have recognised the start of a string literal. Therefore we clear our `StringBuffer` that will hold the content of this string literal and tell the scanner with `yybegin(STRING)` to switch into the lexical state `STRING`. Because we do not yet return a value to the parser, our scanner proceeds immediately.

In lexical state `STRING` another rule demonstrates how to refer to the input that has been matched:

```
[^\n\r\"]+  {  string.append( yytext() ); }
```

The expression `[^\n\r\"]+` matches all characters in the input up to the next backslash (indicating an escape sequence such as `\n`), double quote (indicating the end of the string), or line terminator (which must not occur in a string literal). The matched region of the input is referred to with `yytext()` and appended to the content of the string literal parsed so far.

The last lexical rule in the example specification is used as an error fallback. It matches any character in any state that has not been matched by another rule. It doesn't conflict with any other rule because it has the least priority (because it's the last rule) and because it matches only one character (so it can't have longest match precedence over any other rule).

E.

Change log

i. About CUP Version 0.10

Version 0.10 of CUP adds many new changes and features over the previous releases of version 0.9. These changes attempt to make CUP more like its predecessor, YACC. As a result, the old 0.9 parser specifications for CUP are not compatible and a reading of appendix C of the new manual will be necessary to write new specifications. The new version, however, gives the user more power and options, making parser specifications easier to write.

1. Introduction and Example

This manual describes the basic operation and use of the Java(tm) Based Constructor of Useful Parsers (CUP for short). CUP is a system for generating LALR parsers from simple specifications. It serves the same role as the widely used program YACC [1] and in fact offers most of the features of YACC. However, CUP is written in Java, uses specifications including embedded Java code, and produces parsers which are implemented in Java.

Although this manual covers all aspects of the CUP system, it is relatively brief, and assumes you have at least a little bit of knowledge of LR parsing. A working knowledge of YACC is also very helpful in understanding how CUP specifications work. A number of compiler construction textbooks (such as [2,3]) cover this material, and discuss the YACC system (which is quite similar to this one) as a specific example.

Using CUP involves creating a simple specification based on the grammar for which a parser is needed, along with construction of a scanner capable of breaking characters up into meaningful tokens (such as keywords, numbers, and special symbols).

As a simple example, consider a system for evaluating simple arithmetic expressions over integers. This system would read expressions from standard input (each terminated with a semicolon), evaluate them, and print the result on standard output. A grammar for the input to such a system might look like:

```
  expr_list ::= expr_list expr_part | expr_part
  expr_part ::= expr ';'
  expr      ::= expr '+' expr | expr '-' expr | expr '*' expr
              | expr '/' expr | expr '%' expr | '(' expr ')'
              | '-' expr | number
```

To specify a parser based on this grammar, our first step is to identify and name the set of terminal symbols that will appear on input, and the set of non-terminal symbols. In this case, the non-terminals are:

```
  expr_list, expr_part  and  expr .
```

For terminal names we might choose:

```
  SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD, NUMBER, LPAREN,
and RPAREN
```

The experienced user will note a problem with the above grammar. It is ambiguous. An ambiguous grammar is a grammar which, given a certain input, can reduce the parts of the input in two different ways such as to give two different answers. Take the above grammar, for example. given the following

input:

```
3 + 4 * 6
```

The grammar can either evaluate the `3 + 4` and then multiply seven by six, or it can evaluate `4 * 6` and then add three. Older versions of CUP forced the user to write unambiguous grammars, but now there is a construct allowing the user to specify precedences and associativities for terminals. This means that the above ambiguous grammar can be used, after specifying precedences and associativities. There is more explanation later. Based on these namings we can construct a small CUP specification as follows:

```
// CUP specification for a simple expression evaluator (no actions)

import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner.  */
init with {: scanner.init();              :};
scan with {: return scanner.next_token(); :};

/* Terminals (tokens returned by the scanner). */
terminal            SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal            UMINUS, LPAREN, RPAREN;
terminal Integer    NUMBER;

/* Non terminals */
non terminal            expr_list, expr_part;
non terminal Integer    expr, term, factor;

/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

/* The grammar */
expr_list ::= expr_list expr_part |
              expr_part;
expr_part ::= expr SEMI;
expr      ::= expr PLUS expr
            | expr MINUS expr
            | expr TIMES expr
            | expr DIVIDE expr
            | expr MOD expr
            | MINUS expr %prec UMINUS
            | LPAREN expr RPAREN
            | NUMBER
            ;
```

We will consider each part of the specification syntax in detail later. However, here we can quickly see that the specification contains four main parts. The first part provides preliminary and miscellaneous declarations to specify how the parser is to be generated, and supply parts of the runtime code. In this case we indicate that the `java_cup.runtime` classes should be imported, then supply a small bit of initialization code, and some code for invoking the scanner to retrieve the next input token. The second part of the specification declares terminals and non-terminals, and associates object classes with each. In this case, the terminals are declared as either with no type, or of type `Integer`. The specified type of the terminal or non-terminal is the type of the value of those terminals or non-terminals. If no type is specified, the terminal or non-terminal carries no value. Here, no type indicates that these terminals and non-terminals hold no value. The third part specifies the precedence and associativity of terminals. The last precedence declaration give its terminals the highest precedence. The final part of the specification

contains the grammar.

To produce a parser from this specification we use the CUP generator. If this specification were stored in a file `parser.cup`, then (on a Unix system at least) we might invoke CUP using a command like:

```
java java_cup.Main < parser.cup
```

In this case, the system will produce two Java source files containing parts of the generated parser: `sym.java` and `parser.java`. As you might expect, these two files contain declarations for the classes `sym` and `parser`. The `sym` class contains a series of constant declarations, one for each terminal symbol. This is typically used by the scanner to refer to symbols (e.g. with code such as "`return new Symbol(sym.SEMI);`"). The `parser` class implements the parser itself.

The specification above, while constructing a full parser, does not perform any semantic actions &emdash; it will only indicate success or failure of a parse. To calculate and print values of each expression, we must embed Java code within the parser to carry out actions at various points. In CUP, actions are contained in *code strings* which are surrounded by delimiters of the form `{:` and `:}` (we can see examples of this in the `init with` and `scan with` clauses above). In general, the system records all characters within the delimiters, but does not try to check that it contains valid Java code.

A more complete CUP specification for our example system (with actions embedded at various points in the grammar) is shown below:

```
// CUP specification for a simple expression evaluator (w/ actions)

import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner.  */
init with {: scanner.init();              :};
scan with {: return scanner.next_token(); :};

/* Terminals (tokens returned by the scanner). */
terminal           SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal           UMINUS, LPAREN, RPAREN;
terminal Integer   NUMBER;

/* Non-terminals */
non terminal            expr_list, expr_part;
non terminal Integer    expr;

/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

/* The grammar */
expr_list ::= expr_list expr_part
              |
              expr_part;

expr_part ::= expr:e
              {: System.out.println("= " + e); :}
              SEMI
              ;

expr      ::= expr:e1 PLUS expr:e2
```

```
                {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
                |
                expr:e1 MINUS expr:e2
                {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
                |
                expr:e1 TIMES expr:e2
                {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
                |
                expr:e1 DIVIDE expr:e2
                {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
                |
                expr:e1 MOD expr:e2
                {: RESULT = new Integer(e1.intValue() % e2.intValue()); :}
                |
                NUMBER:n
                {: RESULT = n; :}
                |
                MINUS expr:e
                {: RESULT = new Integer(0 - e.intValue()); :}
                %prec UMINUS
                |
                LPAREN expr:e RPAREN
                {: RESULT = e; :}
                ;
```

Here we can see several changes. Most importantly, code to be executed at various points in the parse is included inside code strings delimited by `{:` and `:}`. In addition, labels have been placed on various symbols in the right hand side of productions. For example in:

```
  expr:e1 PLUS expr:e2
        {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
```

the first non-terminal `expr` has been labeled with `e1`, and the second with `e2`. The left hand side value of each production is always implicitly labeled as `RESULT`.

Each symbol appearing in a production is represented at runtime by an object of type `Symbol` on the parse stack. The labels refer to the instance variable `value` in those objects. In the expression `expr:e1 PLUS expr:e2`, `e1` and `e2` refer to objects of type Integer. These objects are in the value fields of the objects of type `Symbol` representing those non-terminals on the parse stack. `RESULT` is of type `Integer` as well, since the resulting non-terminal `expr` was declared as of type `Integer`. This object becomes the `value` instance variable of a new `Symbol` object.

For each label, two more variables accessible to the user are declared. A left and right value labels are passed to the code string, so that the user can find out where the left and right side of each terminal or non-terminal is in the input stream. The name of these variables is the label name, plus `left` or `right`. for example, given the right hand side of a production `expr:e1 PLUS expr:e2` the user could not only access variables `e1` and `e2`, but also `e1left, e1right, e2left` and `e2right`. these variables are of type `int`.

The final step in creating a working parser is to create a *scanner* (also known as a *lexical analyzer* or simply a *lexer*). This routine is responsible for reading individual characters, removing things things like white space and comments, recognizing which terminal symbols from the grammar each group of characters represents, then returning Symbol objects representing these symbols to the parser. The terminals will be retrieved with a call to the scanner function. In the example, the parser will call `scanner.next_token()`. The scanner should return objects of type `java_cup.runtime.Symbol`. This type is

very different than older versions of CUP's `java_cup.runtime.symbol`. These Symbol objects contains the instance variable `value` of type Object, which should be set by the lexer. This variable refers to the value of that symbol, and the type of object in value should be of the same type as declared in the `terminal` and `non terminal` declarations. In the above example, if the lexer wished to pass a NUMBER token, it should create a `Symbol` with the `value` instance variable filled with an object of type `Integer`. `Symbol` objects corresponding to terminals and non-terminals with no value have a null value field.

The code contained in the `init with` clause of the specification will be executed before any tokens are requested. Each token will be requested using whatever code is found in the `scan with` clause. Beyond this, the exact form the scanner takes is up to you; however note that each call to the scanner function should return a new instance of `java_cup.runtime.Symbol` (or a subclass). These symbol objects are annotated with parser information and pushed onto a stack; reusing objects will result in the parser annotations being scrambled. As of CUP 0.10j, `Symbol` reuse should be detected if it occurs; the parser will throw an `Error` telling you to fix your scanner.

In the next section a more detailed and formal explanation of all parts of a CUP specification will be given. Section 3 describes options for running the CUP system. Section 4 discusses the details of how to customize a CUP parser, while section 5 discusses the scanner interface added in CUP 0.10j. Section 6 considers error recovery. Finally, Section 7 provides a conclusion.

2. Specification Syntax

Now that we have seen a small example, we present a complete description of all parts of a CUP specification. A specification has four sections with a total of eight specific parts (however, most of these are optional). A specification consists of:

- package and import specifications,
- user code components,
- symbol (terminal and non-terminal) lists,
- precedence declarations, and
- the grammar.

Each of these parts must appear in the order presented here. (A complete grammar for the specification language is given in Appendix A.) The particulars of each part of the specification are described in the subsections below.

Package and Import Specifications

A specification begins with optional `package` and `import` declarations. These have the same syntax, and play the same role, as the package and import declarations found in a normal Java program. A package declaration is of the form:

```
package name;
```

where name *name* is a Java package identifier, possibly in several parts separated by ".". In general, CUP employs Java lexical conventions. So for example, both styles of Java comments are supported, and identifiers are constructed beginning with a letter, dollar sign ($), or underscore (_), which can then be followed by zero or more letters, numbers, dollar signs, and underscores.

After an optional `package` declaration, there can be zero or more `import` declarations. As in a Java program these have the form:

Assignment Description

In this programming assignment, you will implement the semantic analysis phases for IC. We expect you to build upon the code that you wrote in the previous programming assignments. You are required to implement the following:

Symbol Tables and Types. Design a hierarchy of symbol tables and a hierarchy of types. Your design should allow each AST node to access the symbol table corresponding to its current scope (e.g. class, method, or block scope), and each entry in the symbol table should have information about the type of the identifier stored in that entry. Any errors which occur during symbol table construction (such as multiply declared identifiers) are considered semantic errors. Your constructed symbol tables should be available to all remaining phases of the compiler. In the rest of your compiler, you will refer to program symbols (e.g., variables, methods, etc.) using references to their symbol table entries, not the actual text.

Semantic Checks. After you have constructed the AST and the symbol tables, your compiler will analyse the program and perform semantic checks. These semantic checks include: (1) scope rules (Section 10 in the IC specification); (2) type-checking rules (Section 15), including a check that the class hierarchy is a tree and checking correct overriding of instance methods in subclasses; (3) checking that the program contains a single `main` method with the correct signature, (4) that `break` and `continue` statements appear only inside loops, (5) that the `this` keyword is only used in instance methods, and (6) that the library class has the correct name (`Library`).

Bonus checks. You are **not** required to check that a local variable is used only after it has been initialized and that a method with a non-void return type returns a value on every control path. If you implement these checks you will get 5 points for each check (be sure to tell us in the documentation that you have implemented these checks).

Error Handling. When your compiler encounters an error it should report information about the error, such as the token and the line number where the error has occurred, or a message describing the violated semantic rule. It is not required to report more than one error; the execution may terminate after the first lexical, syntactic, or semantic error. You must start the error message for semantic errors by `semantic error at line XXX:` and a message of your choice describing the error.

Command line invocation: Your compiler must be invoked with a single file name as argument:

`java` *bin* ⟨*program-filename*⟩ `[options]`

With this command, the compiler will parse the input file, construct the AST and symbol tables, will perform the semantic checks, and will report any error it encounters. Your compiler must also support two command-line options to dump internal information about the AST and the symbol tables:

1. The "`-print-ast`" option: the compiler will print at `System.out` a textual description of the AST for the input file. This is the same print as the one in the previous assignment, with the addition of **type** and **symbol table information** for each AST node.

2. The "`-dump-symtab`" option: the compiler will print a textual description of the symbol tables and the (global) type table. Each entry in the symbol table will be printed on a separate line; the information for each entry should include the name of the identifier, its kind (variable, method, etc.), and its type. Different symbol tables will be separated by blank lines. You must indicate, for each symbol table, what are its children tables. When printing out the type table, print each type on a separate line. For each type include its name, its id, and the id of its superclass if it exists. Order should be primitives

Assignment Description

In this programming assignment, you will implement the translation from the AST of IC programs to the LIR language specified on the web site and validate the translated programs using the microLIR simulator. We expect you to build upon the code that you wrote in the previous programming assignments. You are required to implement the following:

Translation of functions Your compiler should translate all of the functions in the program using the AST and the information gathered during the semantic analysis phase. You compiler should also read the file `libic.sig` when the `-L` option is specified, parse it, and add it to the main AST. The compiler should be able to distinguish between calls to user-defined functions and library functions (in `libic.sig`) and emit the correct call instructions. Of course, the compiler should avoid attempting to emit a translation for the library functions themselves, as their implementation is provided externally.

Class layouts Your compiler should maintain information for each (user-defined) class about its fields and virtual functions. This information is used to determine the offsets for accessing fields (in `MoveField` instructions) and to generate the dispatch tables. In your translation, please print in comments the offset assignment for the fields of each class.
IMPORTANT: When generating field and method offsets for a class, don't forget to account for the fields and methods of its superclass and for overridden methods.

String literals Your compiler should extract all literal strings that exist in the program and translate them to LIR string literals. Instructions using the string literals should be aware of this and use the symbolic names given to the string literals.

Runtime checks You compiler should emit code to check fragile instructions (see LIR documentation) and handlers for runtime errors that print an error message and gracefully terminate the execution. We expect you to implement this code in this assignment.

Command line invocation: Your compiler must be invoked with a single file name as argument:

`java` *bin* ⟨*program-filename*⟩ `[options]`

With this command, the compiler will parse the input file (and `libic.sig`), construct the AST, conduct semantic analysis (reporting any errors it encounters), and translate the program to the LIR language. Your compiler must support the command-line options specified in the previous assignments, as well as the following command line option:

- The "`-print-lir`" option: the compiler will print **into a file**—`file.lir`—the LIR program translated from `file.ic`. The file should be a legal LIR program that can be read and executed on the microLIR simulator giving exactly the same results as we intend the IC program to give.

You are also expected to design a suite of tests (IC programs) that demonstrate the correctness of your translation.

Package Structure: You will implement the IR lowering in a new sub-package `lir`.

What to turn in

You must turn in your code electronically using the instructions posted in the Moodle on the due date, including

Assignment Description

In this assignment, you will generate assembly code from the LIR representation. You will be generating code for Intel's x86 32-bit architecture **using the AT&T syntax**, so all variables and registers will take a full 32 bits of storage. Although it is possible to generate code that stores Booleans more compactly, this data type will be stored in 32-bit words as well. It is recommended that you write comments in the generated code to indicate what instruction sequences correspond to each LIR instruction. You must generate the appropriate code for each of the following:

Instructions in function bodies. Your code should translate each LIR instruction into a sequence of assembly instructions by accessing the parameters/variables/LIR registers using their offsets from the frame pointer (ebp register). To achieve this, your translation should first assign offsets to parameters, local variables, and LIR registers. For parameters, use positive offsets, starting from +8 for the first parameter (which is `this` for virtual functions) and increase by 4 bytes for each parameter. For local variables offsets should start from -4 for the first local variable and decrease by 4 for each variable. LIR registers should be treated as additional local variables (i.e., should have negative offsets starting from the last local variable).

Stack frames. Generate the calling sequences before and after invoking functions, and at the beginning and the end of each function (prologue and epilogue), as discussed in class.

- Registers eax, ecx, and edx are caller-save. You must assume that the contents of caller-save registers may be destroyed at each method call. Registers ebx, esi, edi, ebp, esp are callee-save. Your code should not save/restore these registers, unless you decide to perform register allocation optimizations that work across function calls.
- Function parameters should be pushed on the stack by the caller, in reverse order. That is, the first parameter is pushed last on the stack. In virtual function calls, the first parameter is the receiver object (`this` variable).
- Function calls for static functions and library functions are translated into assembly `call` instructions. Virtual function calls include looking up the address of the function at the given offset in the dispatch table and making an indirect call.
- The return values are always passed in the eax register.
- Pushing and popping the frame. This requires computing the size of each frame, which is the number of bytes needed for the local variables and LIR registers that are stored in the frame. Each function should include a prologue and epilogue section. The prologue section includes the label where the function's instructions start, using the convention `_A_foo` for a function named `foo` in class `A`. The prologue section saves the previous frame pointer, adjust its value and sets a new value for the stack pointer to provide space for the function's local variables and LIR registers. The epilogue should start with a label of the form `_A_foo_epilogue` for a function `foo` in class `A`. This label is used by the translation of `Return` instructions.
- Copying the returned value from eax into the target register for function calls that assign the return value to a register (different from `Rdummy`).

Objects. You must provide support for objects, as discussed in class. This includes generating the dispatch vectors in the data section of the assembly file. Dynamically allocated arrays need not be initialized, because the allocation function fills all of the allocated space with zeros, which corresponds to the default values. Dynamically allocated object fields are also initialized by the `__allocateObject` library function to zero, but your compiler must generate code to properly initialize the DVPtr field (using an assembly instruction `movl $_DV_A,(%ebx)` where %ebx holds the address of the new object).

Arrays and strings. Arrays and strings will be stored in the heap. To create new arrays, use `__allocateArray`; to concatenate strings, use `__stringCat`. String constants will be allocated statically in the data segment. Strings dont have null terminators; instead, each string is preceded by a word indicating the length of the string (see examples on the web-site). Similarly, the length of an array is stored in the memory word preceding the base address of the array (i.e., the location at offset -4). Your translation of the `ArrayLength` instruction retrieves this information.

Runtime checks. You must implement the runtime error handlers `__checkArrayAccess`, `__checkSize`, `__checkNullRef`, and `__checkZero` as assembly instruction sequences. These handlers are not provided as library functions. The runtime handlers must print an error message and gracefully terminate the execution. To implement this, use the code shown in T12. Fragile instructions should be preceded by code that checks erroneous situations and calls the appropriate error handlers (see T12). In addition, each call to a library function should be preceded by code to check that reference type arguments are non-null (e.g., that the arguments of `__stringCat` are non-null references).

Main function. Your main function should be called exactly `__ic_main` (notice the 2 leading underscores) and contain a proper prologue/epilogue sections (like any other function that you translate). This function will be called from an external library function which will take care of passing the command-line arguments to `__ic_main`.

Invoking the Assembler and the Linker

Given an input file `file.ic`, your compiler will produce an assembly file `file.s`. You can then use the assembler to convert this assembly code into an object file `file.o`, and use the linker to convert this object file into an executable file. To run the GNU assembler `as` under the Cygwin environment on `file.s`, use the following command line:

```
as -o file.o file.s
```

To run the GNU linker `ld` on `file.o` and link this object file with the library `libic.a`, use the following command line:

```
ld -o file.exe file.o /lib/crt0.o libic.a -lcygwin -lkernel32
```

The library file `libic.a` is a collection of `.o` files bundled together, containing the code for the library functions that are defined in the language specification, along with runtime support for garbage collection.

Supporting Materials

You can find the library file `libic.a` along with supporting material for this assignment on the course web site. The supporting web page includes documentation for the as assembler; documentation for the x86 instruction set; and several example IC programs along with the corresponding x86 assembly code. The supporting materials also include information for assembling and linking on a Linux environment.

Optional Optimizations

You may choose to implement any of the following optimizations to reduce the number of machine registers used for each function, using the techniques taught in the recitation, and to reduce the number of labels and jump instructions:

Register allocation for single statements (10 pts) The goal of this optimization is to associate LIR registers with machine registers as much as possible, thus avoiding unnecessary memory accesses.

Eliminating unnecessary labels and jumps (5 pts) The goal of this optimization is to: (i) eliminate consecutive labels; (ii) eliminate jump instruction to an immediately following label (i.e., `jmp _label3 _label3:`; and (iii) eliminate labels that are not mentioned by any jump instruction.